„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Wrocław University of Science and Technology

Faculty of Information and Communication Technology

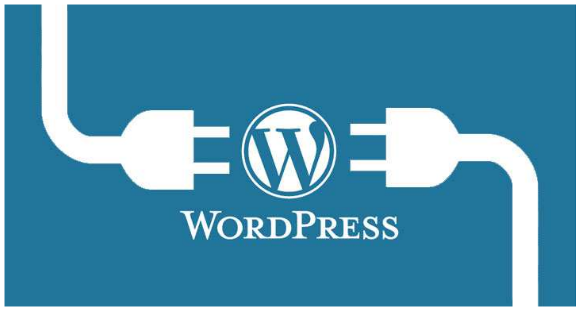
Advanced Web Technologies

Laboratory

Subject:	WordPress plugin implementation
Author:	mgr inż. Piotr Jóźwiak
Date:	february 2025
Time:	2 hours

Table of Contents

Introduction



In the next lab, we will look at how to extend the standard functionality of the WordPress system with a custom plugin. The most important reason to create your own plugin is that it allows you to separate your own code from the WordPress core code. If something goes wrong with our plugin, the rest of the site will generally still function properly. Changing the WordPress source files is probably the worst thing you can do. In addition to the possibility of damaging our system, by doing so we cut ourselves off from the possibility of upgrading to a new version of the system, as the upgrade will automatically erase our code change.

Plugins are simple PHP scripts. As we will see, implementing new functionality into WP is not at all such a difficult process. Of course, at least basic knowledge of the PHP language as well as the functioning of the WordPress system itself is required. In this lab, I assume that the Student already has a basic knowledge of PHP. On the other hand, the rest of the necessary knowledge will be described in the following material.

Wprowadzenie do Pluginów WordPressa

A **WordPress** plugin is a standalone set of code that enhances and extends the functionality of WordPress. Using any combination of PHP, HTML, CSS, JavaScript / jQuery, the plugin can add new features to any part of your site, including the administrative control panel. We can modify the default WordPress behavior or remove unwanted behavior altogether. Plugins allow you to easily customize and personalize WordPress to suit your needs

Since WordPress plugins are self-contained, they do not physically change any of the core WordPress source files. They can be copied and installed in any WordPress installation. An alternative (and strongly discouraged) way to make changes to WordPress is to save new features in the WordPress **functions.php** file, which is stored in the **/wp-include/** folder, or the **functions.php** file, which is part of the theme. This involves a number of potential problems.

WordPress and its themes are regularly updated. If we are not using a child WordPress theme, when **functions.php** is overwritten by an update, the new code will be deleted and we will have to write it from scratch. If we write multiple functions and one of them contains an error that cannot be debugged, we may need to replace the entire file with the original one, removing all the changes made. If our functions are removed from the file, our site will generate PHP errors telling us about the missing functions.

Plugins are never automatically overwritten or removed when installing WordPress updates. If there are coding errors in a plugin, it can usually simply be deactivated in the administrative control panel during repair. If there is a serious error in a plugin, WordPress will sometimes automatically deactivate it, allowing the site to continue operating.

What is WordPress Hook?

WordPress plugins interact with the underlying code by means of so-called Hooks, or certain anchor points. There are two types of hooks.

1. **`Action hooks`** (used to add/remove features)

2. **`Filter hooks`** (used to modify data returned by functions)

Action Hooks

When a user visits any WordPress-based site, in the background the server executes a series of PHP functions (called **actions**) at various call points, which are attached to various anchor points called **Action Hooks**. Using the Action Hooks provided by WordPress, you can add your own functions to the list of actions that are triggered when a certain event is triggered, and you can remove existing functions from anywhere. Action Hooks determine when actions are triggered in the HTML generation process of a page. E.g., before the closing **</head>** tag, an Action Hook named **wp_head()** is called, which in turn calls all actions hooked to **wp_head().**

Action Hooks are contextual - some are invoked on every page of our site, others are invoked only when viewing the Admin Dashboard, and so on. For a full list of available Action Hooks, along with a description of the context in which they are called, please visit Plugin API/Action Reference.

Pinning your own Action function to Action Hook

In order to pin its function to the selected attachment point, the plugin needs to execute a WordPress engine function called **add_action()**. This function expects two parameters. Let's go through the following example to get an idea of how this functionality works:

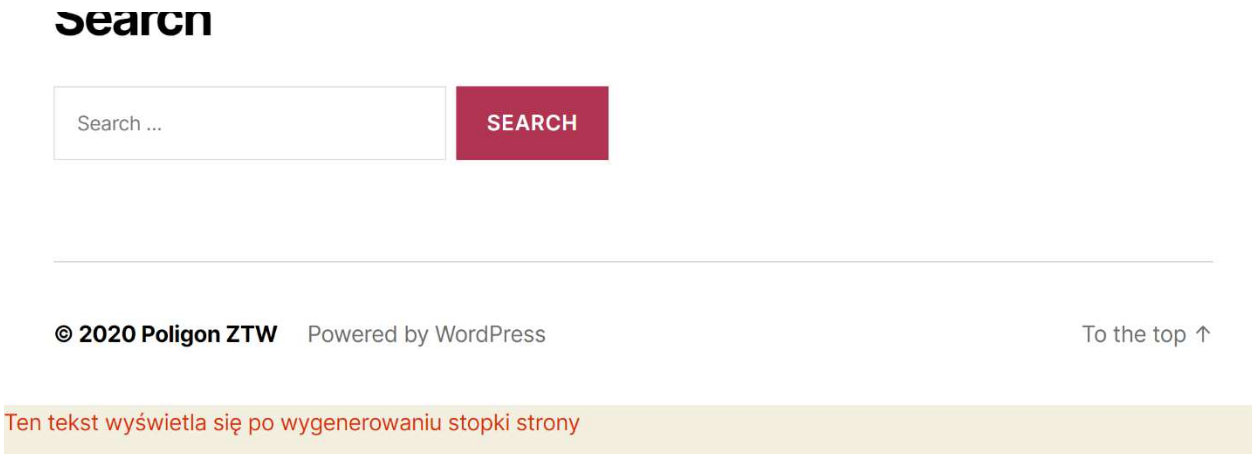
```
// Hook to the 'init' action, which is called after WordPress is finished loading
 the core code
add_action( 'init', 'add_Cookie' );

// Set a cookie with the current time of day
function add_Cookie() {
  setcookie("last_visit_time", date("r"), time()+60*60*24*30, "/");
}
```

- The first required parameter is the name of the Action Hook we want to connect to.
- The second required parameter is the name of the function we want to run.
- The third parameter (optional) is the priority of the function we want to run. We can connect any number of different functions to the same Action Hook. This parameter allows us to rank the actions in the desired order. The default priority is 10, placing the custom function after any of the built-in WordPress functions.
- The fourth parameter (optional) is the number of function arguments. The default value is 1.

The next example is to display some text after generating a standard page footer. To do this, we will pin our action to an Action Hook called **wp_footer()**. This will cause our function to be called whenever the WordPress generator is moments before the **</body>** closing tag. We will pin a function called **mfp_Add_Text()**. The question is where to put our code? For the moment, we will modify the current template for simplicity. We will make the changes in the already mentioned **functions.php** file. To do this example, it is not even required to look into the source files stored on the web server. We will use the editor built into the WordPress admin panel. To open it, let's go to the admin panel. Then, from the side menu, let's choose **Appearance -> Theme Editor**. After a warning about the consequences of modification, the source code editor of the current theme will appear. On the right side, let's search for a file named **functions.php** and select it for editing. At the very top of this file, let's insert the following code:

```
// Define 'mfp_Add_Text'
function mfp_Add_Text()
{
  echo "<p style='color: red;'>Ten tekst wyświetla się po wygenerowaniu stopki st
rony</p>";
}

// Hook the 'wp_footer' action hook, add the function named 'mfp_Add_Text' to it
add_action("wp_footer", "mfp_Add_Text");
```

Let's save the changes and go to the page view. The text was inserted after the footer was generated:



Ten tekst wyświetla się po wygenerowaniu stopki strony

Removal of Action function from Action Hook

To remove an action from the Action Hoop, we need to write a new function that calls the **remove_action()** function. This new function must first be registered, of course. The following example will explain this.

```
// Hook the 'init' action, which is called after WordPress is finished loading th
e core code, add the function 'remove_My_Meta_Tags'
add_action( 'init', 'remove_My_Meta_Tags' );

// Remove the 'add_My_Meta_Tags' function from the wp_head action hook
function remove_My_Meta_Tags()
{
  remove_action( 'wp_head', 'add_My_Meta_Tags');
}
```

Function **`remove_action()`** expects at least two parameters.

- The first required parameter is the name of the Action Hook to which the function is hooked.

- The second required parameter is the name of the function you want to remove.
- The third parameter (optional) is the priority of the original function. This parameter must be identical to the priority that was originally defined when adding the action to the action hook. If you have not defined a priority for your function, do not specify this parameter.

So let's expand our example with the added text in the footer. Let's imagine that we don't want this text displayed on Mondays for some reason. Naturally, the easiest way would be to add the appropriate **if** statement in the body of our Action function. However, we want to see how deleting Action works. Therefore, our example will remove the action on Mondays. Review the code below:

```
// Define 'mfp_Add_Text'
function mfp_Add_Text()
{
  echo "<p style='color: red;'>Ten tekst wyświetla się po wygenerowaniu stopki st
rony</p>";
}

// Hook the 'wp_footer' action hook, add the function named 'mfp_Add_Text' to it
add_action("wp_footer", "mfp_Add_Text");

// Define the function named 'mfp_Remove_TextOnMondays()' to remove our previous
function from the 'wp_footer' action
function mfp_Remove_TextOnMondays()
{
  if (date("l") === "Monday") {
    // Target the 'wp_footer' action, remove the 'mfp_Add_Text' function from it
    remove_action("wp_footer", "mfp_Add_Text");
  }
}

// Hook the 'wp_head' action, run the function named 'mfp_Remove_Text()'
add_action("wp_head", "mfp_Remove_TextOnMondays");
```

In the above example, we registered an action in **wp_head**. Inside this action if today's day is Monday we perform the deletion of another action, adding a caption under the footer. The question is, why did we put our action handling the deletion of another action in **wp_head** instead of **wp_footer**? The answer is fairly obvious. The **wp_head** action is executed earlier than the **wp_footer** action. So we need to ensure that the action is deleted before it executes. This is how we made this arrangement.

Filters and Filter Hooks

A filter function allows you to modify the result data returned by existing functions and must be plugged into one of the **Filter Hooks** - similar to what was done with Action Hooks. Filter Hooks behave similarly to Action Hooks in that they are called at different points in the script and are also contextual. The full list of filter points and the context in which they are called is described in the **WordPress Plugin API/Filter Reference page**.

Pinning Filters

To add a filter function to any Filter Hook, the plugin needs to call a WordPress function called **add_filter()**, with at least two parameters. Review the following example:

```php
// Hook the 'the_content' filter hook (content of any post), run the function named 'mfp_Fix_Text_Spacing'
add_filter("the_content", "mfp_Fix_Text_Spacing");

// Automatically correct double spaces from any post
function mfp_Fix_Text_Spacing($the_Post)
{
 $the_New_Post = str_replace("  ", " ", $the_Post);

 return $the_New_Post;
}
```

- The first required parameter is the name of the Filter Hook.
- The second required parameter is the name of the filtering function we want to run.
- The third parameter (optional) is the priority of the function we want to run.
- The fourth parameter (optional) is the number of arguments, which means how many parameters our filter function can take. The default value is 1.

Plugin Implementation

Now that we have learned the basics of Hooks in WordPress, we can move on to discuss the principles of plugin development. We will discuss how to implement WP extensions based on a sample plugin. We will follow the process of its creation.

The basic operation of our plugin will be to examine whether a given post is a new post (published no more than X days ago). And for such new posts the plugin will generate an appropriate label next to the post title. The plugin will also have a simple admin panel, where we will define what it means that a post is new - in other words, how long a post is treated as new.

Step 1 – naming plugin

We start by coming up with a name for our plugin. The name of the plugin is of little importance since the only recipients will be ourselves. In such a situation, care should be taken that the plugin does not interfere with extensions already installed. However, if we plan to publish our plugin in a public WordPress repository, the name should be unique. The best way to check the uniqueness of the name in a search engine is here: https://wordpress.org/plugins/.

In our case, we will be using the name **Newly Added Post Highliter**.

Step 2 – preparing the folder structure of the plugin

A plugin as we have already mentioned is a set of PHP, CSS, JavaScript and other resourced files. In the simplest approach, a plugin can be a single PHP file. The name of this file should correspond to the name of the plugin. It is best to encapsulate the whole thing in a single parent folder with the name of our plugin.

All plugins are placed in the folder **`wp-content/plugins/`**. So let's prepare our plugin folder structure as follows:

```
NewlyAddedPostHighliter
├── css
│   └── style.css
├── image
├── js
└── newly-added-post-highliter.php
```

This is, of course, only a certain suggestion. We do not have to strictly adhere to such a hierarchy, if another one suits our needs better, it is safe to change it. In the above example, I also put two files right away. The first **newly-added-post-highliter.php** will be the main file of our plugin. While the **css/style.css** file, as the name suggests, will define the layout.

Step 3 –plugin meta information

We put the description of our plugin in the main PHP file as a comment. It requires the use of a specific format. Based on this comment, the WordPress engine loads the necessary information about the plugin. You can read more about the requirements for this header here: https://developer.wordpress.org/plugins/plugin-basics/header-requirements/. In our case, we will insert the following in the newly-added-post-highliter.php file:

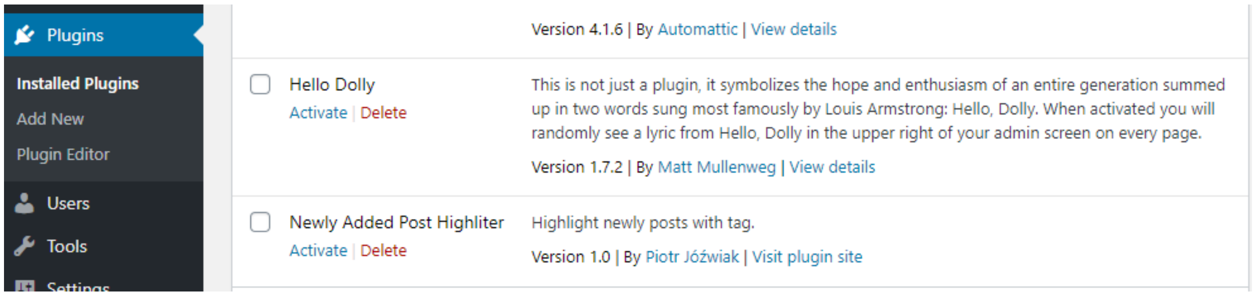
```php
<?php
/**
 * Plugin Name:       Newly Added Post Highliter
 * Plugin URI:        https://example.com/plugins/Newly Added Post Highliter/
 * Description:       Highlight newly posts with tag.
 * Version:           1.0
 * Requires at least: 5.0
 * Requires PHP:      7.2
 * Author:            Piotr Jóźwiak
 * Author URI:        https://darksource.pl/
 * License:           GPL v2 or later
 * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
 */
```

The necessary minimum is to enter the Plugin Name field. However, it is worth including other fields here as well.

Step 4 – launching the plugin

Before writing any code, it is worth running the plugin on the website. Since it doesn't actually have even a single line of code yet, it shouldn't cause errors. So we are assured that nothing bad will happen to our site after running it. We will develop the plugin incrementally, so that we can see which code causes script errors.

So, let's upload the **NewlyAddedPostHighliter** folder and its contents to the **wp-content/plugins/** folder on the web server. Next, let's log into the WordPress dashboard and go to the **Plugins** section. In the list of plugins, we should see our new plugin:



Version 4.1.6 | By Automattic | View details

Hello Dolly
Activate | Delete

This is not just a plugin, it symbolizes the hope and enthusiasm of an entire generation summed up in two words sung most famously by Louis Armstrong: Hello, Dolly. When activated you will randomly see a lyric from Hello, Dolly in the upper right of your admin screen on every page.

Version 1.7.2 | By Matt Mullenweg | View details

Newly Added Post Highliter
Activate | Delete

Highlight newly posts with tag.

Version 1.0 | By Piotr Jóźwiak | Visit plugin site

Let's activate it by clicking on the link ***Activate***.

Step 5 – implementation of plugin functionality

We will start the implementation of our plugin by placing the styles. Since it is not the purpose of this lab to discuss the principles of CSS, so just for the sake of completeness of the example, I present the contents of the **css/style.css** file. Let's remember that when we write our own plugin, we should put the definition of our own styles corresponding to the graphical assumptions of the site. For the purposes of this example, this file has the following content:

```css
.naph_marker{
    display:none;
}


.naph_marker{
    color: white;
    margin-left: 10px;
    background-color: red;
    padding: 1px 10px;
    border-radius: 5px;
    font-size: 0.5em;
    font-style: italic;
}

article .naph_marker{
    display: inline;
}
```

So let's move on to writing the plugin itself. All the code will be contained in the main PHP file. We will discuss the code in the order of its placement.

Implementation of the admin panel

The first functionality we will take care of will be to write an admin panel, where we will set the parameters of our plugin. Configuration will be very simple and will boil down to setting one parameter specifying the number of days from publication for which a given post will be marked with our tag.

Let's start by registering the link in the Settings menu. To do this, we insert the following code in the main file:

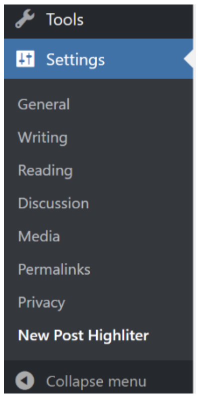
```
function naph_admin_actions_register_menu(){
    add_options_page("Newly Added Post Highliter", "New Post Highliter", 'manage_
options', "naph", "naph_admin_page");
}

add_action('admin_menu', 'naph_admin_actions_register_menu');
```

This simple example performs two steps. In the first step, it registers an Action Hook on the **admin_menu** event. This is the event that is executed when the admin menu is generated. We attach a function named **naph_admin_actions_register_menu** to this event. This function calls the **add_options_page** function. This function is responsible for adding menu items in the **Settings** section. You can read the details of this function here: https://developer.wordpress.org/reference/functions/add_options_page/. It accepts up to 6 parameters:

- The first parameter is Page Title.
- The second parameter is responsible for the name/text displayed in the admin menu.
- The third parameter is the required authorization for that element, known as ***capability***. You can read more about roles and permissions here: https://wordpress.org/support/article/roles-and-capabilities/. In our case, we insert the required permission to change settings stored in a capability named: **manage_options**.
- The fourth parameter is a string that will build the URL to the administration page. It must be unique throughout the site.
- The fifth parameter is the name of the function to be run to generate the admin panel page.

Since we have already registered a reference to the function that generates the admin panel, let's save a simplified version of it for now, so that we can test the operation of our code. In our case it will look like this (we will still modify it):

```
function naph_admin_page(){
?>
    <h1>Newly Added Post Highliter</h1>
<?php
}
```



Let's save the changes to the file and refresh the admin panel page. Let's click on the Settings section. Now a link to our admin panel should be generated.

When you click on it, you will see a blank page with the header saved in the function that generates the page content.

So let's move on to writing the function that handles the admin panel. In our case, it looks as follows:

```
function naph_admin_page(){
  // get _POST variable from globals
  global $_POST;

  // process changes from form
  if(isset($_POST['naph_do_change'])){
    if($_POST['naph_do_change'] == 'Y'){
      $opDays = $_POST['naph_days'];
      echo '<div class="notice notice-success is-
dismissible"><p>Settings saved.</p></div>';
      update_option('naph_days', $opDays);
    }
  }

  //read current option value
  $opDays = get_option('naph_days');

  //display admin page
?>
  <div class="wrap">
    <h1>Newly Added Post Highliter</h1>
     <form name="naph_form" method="post">
       <input type="hidden" name="naph_do_change" value="Y">
       <p>Highlight post title for
       <input type="number" name="naph_days" min="0" max="30" value="<?php echo $
opDays ?>"> days
       </p>
       <p class="submit"><input type="submit" value="Submit"></p>
     </form>
  </div>
<?php
}
```

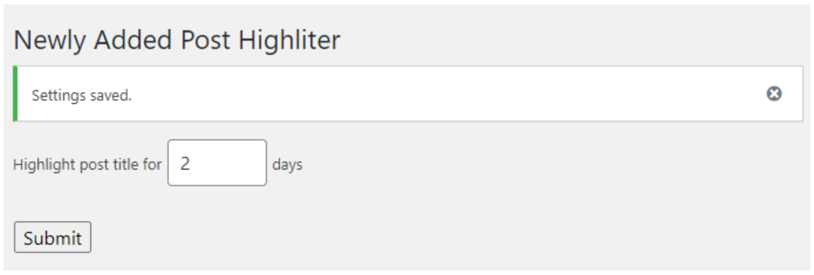
As you can see from the comments on the code, this function takes care of several things in succession. The first part is responsible for processing the form and saving the new data to the WordPress database. For this we use the built-in functionality of the CMS engine in the form of two functions:

- **update_option**(string $option, mixed $value, string|bool $autoload=null)
- **get_option**(string $option, mixed $default = false)

The first one stores the value under the identifier **naph_days** in the database. We don't have to worry about how to store this data. WordPress takes care of that for us on its own. The second function loads the current value of this option. As you can see, this is a very easy and pleasant to use solution.

The next part of the discussed function generates a form with the current settings.

That's all the functionality responsible for handling the administration of the plugin. Let's save the changes and go to the browser. Let's try to set a value for our option and test if everything saves. The result should resemble the one in the image below:


Newly Added Post Highliter

Settings saved.

Highlight post title for 2 days

Implementation of front-end functionality

It's time to write the part of the plugin responsible for generating a marker for the titles of the latest posts. To do this, we need to write the code that handles this requirement into the plugin's main file. You can add it at the end of this file. In our example, this code looks like this:

```
function naph_mark_new_post_title($content, $id){
    //read post publish date
    $date = get_the_date('Ymd', $id);
    //get current date
    $now = date('Ymd');
    //get setting for how long post is a new post
    $opDays = get_option('naph_days');
    //generate proper post title
    if($now - $date <= $opDays)
        return $content."<sup class=\"naph_marker\">new</sup>";
    return $content;
}

add_filter("the_title", "naph_mark_new_post_title", 10, 2);
```

As before, we register the **naph_mark_new_post_title** function to a filter named **the_title**. This filter is responsible for displaying the post title. We will modify it depending on the settings of our plugin.

The marking logic is implemented in the **naph_mark_new_post_title** function; this function receives two parameters:

- The first is the current value of the post title,
- The second is the post ID.

First, we load the post publication date information with the **get_the_date** function, passing its format and our post ID. Then we load the actual date. Next, we load our plugin settings.

At the end, we generate the post title with a marker if the post is fresher than the number of days loaded from the plugin settings or the title with no change.

Let's save the changes and check if the plugin works. Of course, for this moment we have not yet connected the styles. Therefore, the effect is not very impressive. We will deal with it in the next step.

Style file registration

The last thing we need to do is to register the previously defined styles. We also do this in our main PHP file. At the very end, we add the following code:

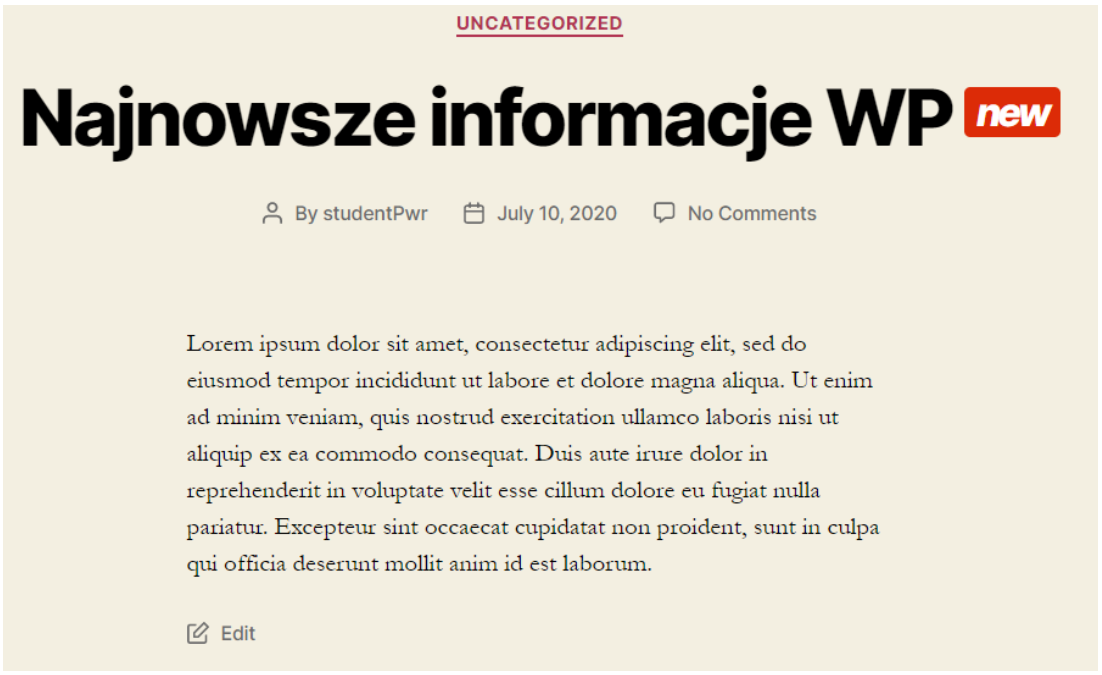
```
function naph_register_styles(){
    //register style
    wp_register_style('naph_styles', plugins_url('/css/style.css', __FILE__));
    //enable style (load in meta of html)
    wp_enqueue_style('naph_styles');
}

add_action('init', 'naph_register_styles');
```

For this, we will again use an action pin to the Action Hook called **init**. The **init** action is triggered when the WordPress engine loads, but before any HTTP header is sent. In the pinned function, we use two functions to:

- **`wp_register_style()`** – registers our style file with the name from the first argument,
- **`wp_enqueue_style()`** – indicates to append the file registered under the name from the argument to the html of the page.

The above mechanism gives full control over style management. By separating this functionality into registration and queue, we have the ability to attach style files (as well as script files) only in the required places. We do not have to load these files always - thus saving transfer and speeding up the operation of the website.

Let's save the changes to the file. It's time to check how the plugin works. Let's go to the browser. As a result, we should get a result similar to the following:



UNCATEGORIZED

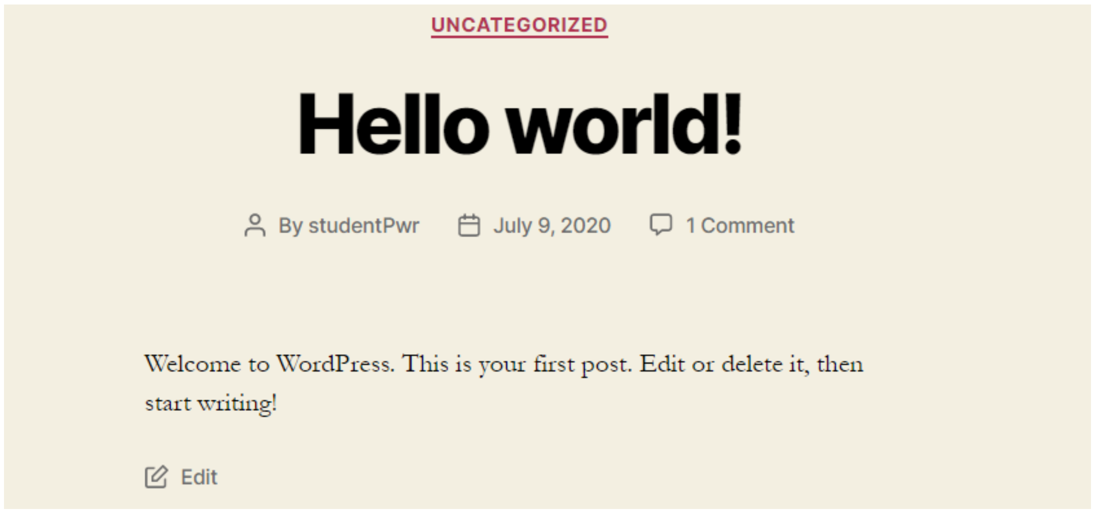
Najnowsze informacje WP new

By studentPwr July 10, 2020 No Comments

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Edit

However, for the older post, the title remained unchanged:



UNCATEGORIZED

Hello world!

By studentPwr July 9, 2020 1 Comment

Welcome to WordPress. This is your first post. Edit or delete it, then start writing!

Edit

Debugging WordPress

Just one more digression at the end of this lab. When writing any code, it is something natural that errors will occur. WordPress will not display error messages by default to protect this sensitive information from attackers. However, during implementation, they are an invaluable aid to the developer.

To run this information just add in the file **`wp-config.php`** the following line of code:

```
define( 'WP_DEBUG', true );
```

From now on, error messages should be much more helpful for troubleshooting. More on debugging WordPress can be found here: https://wordpress.org/support/article/debugging-in-wordpress/.